OUTOPIA: Private User Discovery on the Internet

Panagiotis Papadopoulos
FORTH-ICS/University of Crete, Greece

Michalis Pachilakis
FORTH-ICS/University of Crete, Greece

Antonios A. Chariton
University of Crete, Greece

Evangelos P. Markatos
FORTH-ICS/University of Crete, Greece

ABSTRACT

Before being able to communicate with one another over the Internet, users in messaging applications need to *discover* each other and learn their IP addresses. Today, this *User Discovery* process is closely coupled with the communication provider. As a result, these providers are able to find *(i) who is talking to whom, (ii) who is friends with whom and (iii) where is everybody located in the Internet address space at any time, even when there was no communication channel ever established*, positioning this way themselves as powerful "Big Brothers".

In this paper, we show that it is easy for friends to discover each other without the need of a centralised service provider that monitors each and every move they make. We propose *OUTOPIA*: a system to provide privacy-preserving User Discovery on the Internet. With OUTOPIA, users are able to discover each other, without revealing their social connections. We implemented a prototype of our approach and showed that it is inherently scalable, able to handle tens of thousands of users per server. Our preliminary performance results suggest that users are able to discover each other in no more than a few milliseconds, while generating negligible traffic overall.

CCS CONCEPTS

• **Security and privacy** → **Privacy protections**; **Pseudonymity, anonymity and untraceability**.

ACM Reference Format:
Panagiotis Papadopoulos, Michalis Pachilakis, Antonios A. Chariton, and Evangelos P. Markatos. 2022. OUTOPIA: Private User Discovery on the Internet. In *15th European Workshop on Systems Security (EUROSEC '22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3517208.3523756

1 INTRODUCTION

The first step of every single application on the Internet includes an *IP address discovery process*: e.g., before a browser can download a single byte from a web server, it first needs to find the server's IP address. These IP addresses are currently provided by the DNS infrastructure: a third-party infrastructure, independent from *any* individual server or client, which *translates* domain names to IP addresses. The independent, distributed nature of DNS enables flexibility, reliability, and makes it practically impossible for any individual organisation to control the domain name translation process.

The exact same discovery process is needed when user needs to contact another user: although two friends may know each other by name, in order for them to communicate over the Internet, they need to know each other's IP addresses. However, when web servers' IPs do not change (or change sporadically), the mobile users' IP addresses change quite frequently: when users go to work, when they return home, when they connect to a free WiFi in a restaurant, etc. Consequently, in user-to-user communication applications (e.g., VoIP, chat, file/image/video sharing), the User Discovery process constitutes an essential component for users to *translate* the username of their friends to IP addresses of that very moment.

Contrary to the well-established discovery mechanism of the Internet (i.e., the DNS), in user-to-user applications there is *no independent, third-party IP address translation infrastructure*. Instead, User Discovery is bundled with the communication process [1–3] in a single service that enables users to both discover and communicate with each other. To make matters worse, contrary to the DNS paradigm, users cannot query only for the friend they plan to connect with. Instead, they have to upload their phone's entire address-book [4] in order for the service to (i) check if their friends are users of the app and (ii) what is their current IP address, thus compromising the privacy of their entire social graph. This means that users *reveal phone numbers of every single person they are socially connected with, even if they do not ever communicate with them through the app, even if these friends are not registered with the specific app!* [5, 6]

As a consequence, although the mentioned popular messaging apps offer practical user discovery and encrypted conversation contents, they have the power to collect a wealth of metadata including (i) *which user is talking to whom*, (ii) *which user is friends with whom*, and (iii) *what is the IP address of all users at all times* without needing the users to exchange *any* message. The information extracted from such metadata can be later sold [7–9] or included as an asset in the company's future buy-out [10, 11]. Recent revelations show [12] that friend-lists and address-books are sought after by Intelligence Agencies [13, 14]; by analysing a user's social graph one can reveal sensitive information such as interests, political or religious beliefs, sexual preferences [15], etc.

Building a private User Discovery service that can protect metadata while being scalable, able to sustain an ever-increasing number of users without imposing high overheads has been proved to be extremely challenging [16]. In some cases, it can even become an insurmountable obstacle to wider adoption of important communication services (e.g., PGP) [17]. Anonymity networks that allow

EUROSEC '22, April 5–8, 2022, RENNES, France
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9255-6/22/04…$15.00
https://doi.org/10.1145/3517208.3523756

Approach	Purpose	SOOBC	Assumptions	Scalability	Overhead per user (1M users)
Apres[19]	1-1 Messaging	yes	Tor immune to traffic analysis	scalability of Tor	(not evaluated)
Pung[22]	1-1 Messaging	yes	-	32K active users per 4-server setup	1,333 Kbps
Vuvuzela[18]	1-1 Messaging	yes	at least one non-compromised server	2M users	96 Kbps
DP5[23]	Chat Presence	yes	at least one non-compromised server	1M users	18.00 Kbps
PROUD[27]	User Discovery	yes	non-colluding nodes	millions of users	4.8 Kbps
OUTOPIA	User Discovery	yes	honest but curious nodes	millions of users	2 Kbps

Table 1. Comparison of related approaches. Some rely on Tor and some are based on non-compromised servers. Their overhead varies from a few Mbps down to several tens of Kbps. Interestingly, despite their differences, they all have a common assumption: all of them assume the existence of a secure out-of-band channel (SOOBC) that enables them to bootstrap their process. This channel allows friends to exchange an one-time-only boot-strapping information: a public key, a password, or whatever each method is using.

anonymous messaging, e.g., Vuvuzela [18], Apres [19], Riposte [20], Dissent [21]), Pung [22], DP5 [23], Talek [24] or AnoNotify [25] can provide metadata privacy but either do not consider User Discovery as part of their design or rely on Tor, PIR[26]S or anonymous broadcasts (see Table 1). As a consequence, their limited scalability and heavy overheads open questions about their practicality.

In this paper, we advocate that similar to the world wide web paradigm, User Discovery needs to be independent and unbundled from encrypted messaging apps, enabling users to find each other without relinquishing the privacy of their entire address book. Therefore, we propose *OUTOPIA*: an Oblivious User-TO-Personal IP Address discovery service to let users find the IP address of their friends in a privacy-preserving way. What is more, OUTOPIA is lightweight, able to provide *measurable privacy* guarantees to every social association of the user. So users are aware of the exact probability for each of their friendships to be revealed.

The contributions of our approach can be summarised as follows:

(1) We design *OUTOPIA*: an independent third-party discovery service to provide privacy-preserving User Discovery without revealing who is socially connected with whom. OUTOPIA, ensures the confidentiality of the stored data (user-to-IP mappings) and provides measurable privacy to the associated metadata (who queries for whom).
(2) We present an analytical evaluation of our approach in terms of privacy and we derive closed-form formulas for the disclosure probability and the sizing of the buckets used in our system. By conducting a simulation-based analysis we present the generated traffic as a function of the provided level of privacy in our system.
(3) We implement a prototype of OUTOPIA and early performance results show that it imposes negligible latency while at the same time it provides measurable privacy to the user's social graph.

2 THREAT MODEL

In this paper, we assume a TLS-capable directory server, which holds information about the (encrypted) IP addresses of its users. Users (i) inform the server about their IP address (i.e., `set` operation), (ii) inform the server about any changes in their IP address (i.e., `update` operation), and (iii) ask the server for the IP addresses of their friends (i.e., `get` operation). All IP addresses above are stored and transferred encrypted. The server can be maintained by anyone as per the DNS and Tor paradigm.

Similar to other studies [27], we assume this directory server is (i) *honest*: it faithfully implements the proposed protocol without trying to cheat by performing active attacks (e.g., by injecting dummies) against the service it hosts, thus jeopardising its own users. But (ii) it is also *curious* aiming to collects all the data it is given and tries to infer as much information as possible. The server may use this information to, for example, discover who is friends with whom and reconstruct its users social graph: e.g., if Bob updates its IP address and later Alice reads this update, the server may infer that Alice is friends with Bob. Additionally, we assume the server being a potential victim of data leak due to either breaches [28], bad maintenance [4, 29, 30], or even *insider attacks*.

Bootstrapping: Similar to related works (see Table 1) but also current apps (e.g., WhatsApp, Telegram), we assume that the users bootstrap their friendship, by exchanging necessary information (e.g., public keys, bucket names) via a secure out-of-band channel.

Notation	Explanation
dd_{Alice_Bob}	The dead-drop used by Bob to communicate his IP address to Alice
$name(dd_{Alice\_Bob})$	The name of the dead-drop used by Bob to communicate his IP address to Alice
B_{Alice_Bob}	The bucket where dd_{Alice_Bob} is placed
I_{Alice_Bob}	The index inside bucket B_{Alice_Bob} where dd_{Alice_Bob} is placed
T	Duration of an epoch
N	Average number of friends per user
TB_{Alice_Bob}	Timestamp of the latest version of B_{Alice_Bob} Alice has downloaded.
U	Update rate of IP addresses
DP	Disclosure Probability.

Table 2. Summary of Notation

3 SYSTEM OVERVIEW

In the rest of the paper, we engage two users, called by convention Alice and Bob and a TLS capable server that provides User Discovery. Let us assume that Alice is friends with Bob and she would like to talk to him. Assume also that Bob would like to accept Alice's communications and thus he would like to confidentially share his IP address. This way, Alice can open a socket to Bob's device and communicate in a peer-to-peer fashion.

The main concept of our approach is the concept of a dead-drop: a message with a long hard-to-guess name that contains the encrypted IP address of Bob. The dead-drop is what Bob uses to place his IP address without needing any authentication: the long hard-to-guess name of the dead-drop is Bob's assurance that nobody else updates the dead-drop. The dead-drop name is shared between Bob and Alice during their friendship's bootstrapping. To hide the fact that Alice and Bob are accessing the same dead-drop we put the dead-drops in buckets. Although Bob updates his own dead-drop only, Alice reads the entire bucket where Bob's dead-drop is stored. The exact size of the bucket is a parameter of OUTOPIA: the larger the bucket size, the less certain the server is that Alice would like actually to query for Bob. For example, if the size of the bucket is set to be 1,000 dead-drops, Alice may be friends with any of the 1,000 people who created these dead-drops. In our terminology, the disclosure probability of Alice and Bob's friendship will be 0.1% (see Section 4.1 for detailed privacy analysis).

In Figure 1, we present the high level overview of our design, where a OUTOPIA server is responsible for (i) receiving Bob's dead-drops and placing them in buckets and also (ii) respond to Alice by sending her the bucket she queries for. When Bob changes his own IP address, (step 1a), encrypts it (step 2) puts all the information in a dead-drop and sends the dead-drop to a OUTOPIA server (step 3). The red arrow updates the dead-drop that is for Alice and the rest of the arrows update the dead-drops for the rest of Bob's friends. The OUTOPIA server adds the dead-drop in bucket 8 (step 4) and gives this information to Bob who conveys the information back to

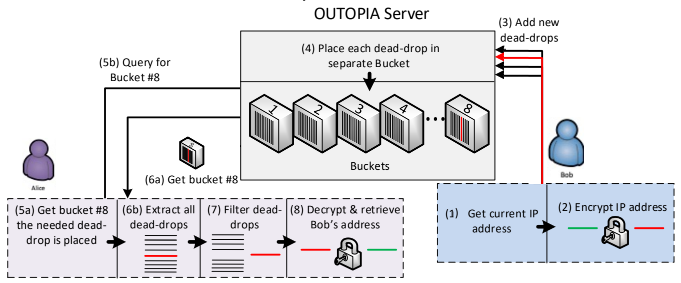
OUTOPIA Server
(3) Add new dead-drops
(4) Place each dead-drop in separate Bucket
(5b) Query for Bucket #8
1
2
3
4
8
Buckets
(6a) Get bucket #8
Alice
Bob
(5a) Get bucket #8 the needed dead-drop is placed
(6b) Extract all dead-drops
(7) Filter dead-drops
(8) Decrypt & retrieve Bob's address
(1) Get current IP address
(2) Encrypt IP address

Fig. 1: Design overview: Bob encrypts his IP address (step 2) and creates one dead-drop per friend before sending them to OUTOPIA (step 3). OUTOPIA adds the dead-drops in buckets (step 4) and responds back to Bob with the bucket numbers. When Alice wants to find Bob's current IP address, she asks OUTOPIA for the proper bucket (bucket #8 in the example - steps 5a and 5b). The server responds with the bucket (step 6a) to Alice, who filters out the dead-drop she is not interested in (steps 6b and 7). Finally Alice decrypts the contents of Bob's dead-drop (step 8) and finds out his IP address.

Alice. When Alice would like to find Bob's current IP address, she asks a OUTOPIA server for bucket 8 (steps 5a and 5b). The server provides the bucket (step 6a) and Alice filters out the dead-drop she is interested in (steps 6b and 7). Finally, Alice decrypts the contents of the dead-drop (step 8) and finds out Bob's IP address.

3.1 Protocol description

Let us describe the basic operation of OUTOPIA in more detail using the notation as summarised in Table 2. OUTOPIA protocol consists of three main operations:

1. Add a new dead-drop: When Bob wants to let Alice know of his current network address, he creates a new dead-drop (say dd_{Alice_Bob}) in the form of a key-value pair, with the dead-drop name (say $name(dd_{Alice\_Bob})$) as key and the encrypted IP address as value. Bob has one dead-drop for each and every one of his friends. Thus, no two friends of Bob share the same dead-drop and thus he can individually *unfriend* Alice. The name of each dead-drop is unique, generated by the user using SHA-256 function and therefore it cannot be guessed (nor overwritten). Apart from Bob, nobody else can learn the name of his dead-drop. Regarding the dead-drop's payload, it includes Bob's current IP (IP_{Bob}) encrypted with Alice's public key K^{pub}_{Alice} (only she can decrypt): $E_{K^{pub}_{Alice}}(IP_{Bob})$. As soon as the server receives Bob's new dead-drop request, it stores the received tuple:

dead-drop name:	dead-drop value:
$h(name(dd_{Alice\_Bob}))$	$E_{K^{pub}_{Alice}}(IP_{Bob})$

Note that the server performs another round of hashing before storing $name(dd_{Alice\_Bob}$. By doing that, we ensure that even in the case of a data breach, the attacker cannot learn the name of the dead-drop and this way impersonate Bob by overwriting his dead-drop. The server stores the above tuple in a bucket. Once the bucket is full[1], the server informs Bob of (i) the bucket name (i.e., (B_{Alice_Bob})) as well as (ii) the index (position) of the added dead-drop in it (i.e., I_{Alice_Bob}). Bob share this information with Alice during the bootstrapping process of their friendship. The server makes sure that all dead-drops are distributed to the buckets in a way that no bucket receives more than one dead-drop from the same user. Indeed, if Bob had several of his dead-drops in the same bucket, then the server might conclude with high probability that Alice is friends with Bob.

2. Query for a dead-drop: When Alice wants to find the IP address of Bob, she queries the server for the *entire* bucket B_{Alice_Bob}: the bucket which contains several dead-drops, including the dead-drop Bob created for Alice. The server responds with all the encrypted IP addresses in the bucket and their relative position in this bucket (but not the hashed names of their dead-drops). Once she receives bucket B_{Alice_Bob}, Alice selects Bob's dead-drop (position I_{Alice_Bob}) and decrypts its contents by using her private key.

Time and frequency of querying is another piece of metadata that may reveal social association. When Bob changes his IP address, Alice (possibly along with several other people who may not be friends with Bob) may ask for bucket B_{Alice_Bob}. When Bob changes his IP address again, Alice may again ask for bucket B_{Alice_Bob}. Pretty soon, the server might be able to establish a pattern: When Bob changes his IP address, Alice asks for bucket B_{Alice_Bob}, thus, Alice probably is friends with Bob.

To deprive the server from this knowledge, we divide the time into periods – *epochs*. Each *epoch* lasts for T seconds. All clients of OUTOPIA (including Alice) at each and every epoch request all buckets they are interested in. That is, periodically, all clients ask for all buckets that contain dead-drops of their friends. To make this possible while reducing the amount of information transmitted, in OUTOPIA, the server does not respond to queries with the *entire* bucket B_{Alice_Bob} to Alice, but only its changes (i.e., the *Delta*) since the last time she queried for it. This way, Alice periodically queries for bucket deltas using again (a) the bucket ID (e.g., B_{Alice_Bob}) and (b) timestamp $T_{B_{Alice\_Bob}}$ of the most recent bucket version she has downloaded. If the server has a later version (timestamp larger than $T_{B_{Alice\_Bob}}$) then it responds with the changes (i.e., only the modified dead-drops), otherwise with a NO-CHANGES message.

3. Update an existing dead-drop: It is apparent, that whenever Bob changes his IP address he needs to update the contents of dd_{Alice_Bob} so Alice can relocate him in the network. To do that, he issues an update request to the server denoting the dead-drop name (i.e., $name(dd_{Alice\_Bob})$) that he wants to update, along with the new value (i.e., his encrypted IP address). The server computes the hash function $h(name(dd_{Alice\_Bob}))$ and updates the dead-drop. It is possible that at some point in time Bob may want to *unfriend* Alice. This implies that Alice should not be able to know Bob's current IP address any more. To do so, whenever Bob wants to drop Alice from a friend, he updates the contents of dead-drop dd_{Alice_Bob} to a value that is not his valid IP address. In this way, Alice will not be able to talk to Bob again, since she does not know his IP address any more.

4 ANALYTICAL EVALUATION

4.1 Performance Analysis

We analyse the traffic generated by OUTOPIA. Let us assume that the duration of the epoch is T (units of time), and each user has (an average of) N friends. For simplicity, we assume that each friend of Alice is placed in a different bucket. In order to have the most up-to-date IP addresses of all her friends, Alice will at each and every epoch, receive updates from N buckets. The overhead of these updates includes:

(i) a constant overhead (say C_0), which is independent from the number of dead-drops that have changed in the bucket. This constant overhead has to be paid once per epoch (per bucket) independently from how many dead-drops have changed, and

[1]The bucket needs to be full before Alice is allowed to access it, otherwise the server might be able to infer that she is friends with Bob (with higher-than-normal probability.

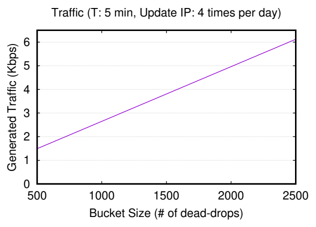

Fig. 2: Traffic generated per client as a function of bucket size. We see that the traffic increases linearly with the size of the bucket as expected. For a bucket size of around 1000 dead-drops OUTOPIA generates about 2.5 Kbit/s, which is reasonably small for today's networks.

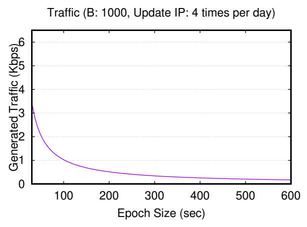

Fig. 3: Traffic generated per client as a function of the duration of the epoch T. We see that the traffic decreases inversely proportional to the duration of the epoch. It is interesting to see that for epoch size of around 300 seconds, OUTOPIA generates less than 0.5 Kbit/s.

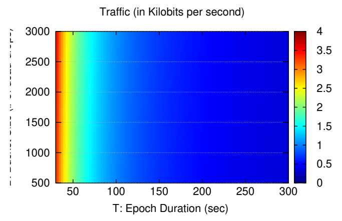

Fig. 4: Traffic generated as a function of bucket size B and epoch duration T. We see that for the largest set of parameters the Traffic generated is between 0.5 Kbit/s and 1 Kbit/s (less than the DNS-based approach [27]).

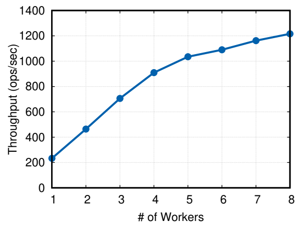

Fig. 5: Throughput of a bucket read operation. The throughput of the server increases near linearly with the more clients querying the server. After 1,200 ops/sec, the performance levels off, as the network is becoming saturated with data transfers.

(ii) an overhead proportional to the number of changed dead-drops. Assuming an update rate of U (per unit of time), then we expect $U \times B \times T$ users to have changed their dead-drop in bucket B. This will result in an amount of data (less or) equal to $C_1 \times U \times B \times T$, where C_1: constant factor related to the size of a dead-drop.

Thus, for each bucket, each user will receive $(C_0 + C_1 \times U \times B \times T)$ amount of data per epoch or $(C_0/T + C_1 \times U \times B)$ amount of data per unit of time. If each user has an average of N friends, and since we assume that each and every friend's dead-drop will be in a different bucket, the total amount of data received (per unit of time) will be:

$$(C_0/T + C_1 \times U \times B) \times N \tag{1}$$

The impact of the Bucket Size: To get a feeling of how much traffic is generated we make some reasonable assumptions about the parameters involved. Let us assume a constant overhead of 64 bytes (i.e., $C_0 = 64$), an incremental overhead of 32 bytes per dead-drop (i.e., $C_1 = 32$), an update rate (U) of 4 times a day (1: home wifi-cellular, 2: cellular-office wifi, 3: office wifi-cellular, 4:cellular-home wifi), and an average friend population of around 200 people (i.e., $N = 200$). Figure 2 shows the traffic generated by OUTOPIA per user per unit of time as a function of the bucket size B for the above defined values of the rest of the parameters. We see that the traffic increases linearly with the size of the bucket as expected. It is interesting to see that for a bucket size of around 1000 dead-drops, OUTOPIA generates about 2.5 Kbit/s, which is reasonably small for today's networks. Varying other parameters of equation 1, one may easily compute OUTOPIA's overhead under different circumstances. The main point, however, here is that for reasonable sets of parameters, the overhead remains relatively low.

The impact of the Epoch Duration: In Figure 3, we show the traffic generated by OUTOPIA as a function of the duration of the epoch T. We see that for an epoch size of around 200-400 seconds, the generated traffic is less than 0.5 Kbit/s, which is practically negligible for today's networks. In Figure 4, we vary both T (the duration of the epoch in x axis), and B (the bucket size). We see that for the most part, the generated traffic stays in the range of 0.5 to 1 Kbit/s (shown in blue colour), which is practically insignificant.

4.2 Privacy Analysis

The Disclosure Probability: As a metric of privacy, we use the notion of *disclosure probability* as introduced in [31, 32]. As disclosure probability (DP), we define the probability of Alice and Bob's association to be revealed to the server (i.e., the confidence of the server that Alice is friends with Bob).

A Priori and a Posteriori probabilities: To place this work in the proper context, we assume that the server has some *a priori* knowledge about Alice and Bob. This knowledge enables the server to compute an *a priori* probability of whether Alice is friends with Bob. This probability can be very low, or very high. For example, if the server has no information about Alice and Bob, the *a priori* probability will be low. On the other hand, if the server knows e.g., that Alice is married to Bob, then the *a priori* probability will be high. The main point here, is that in all cases, the server has some *a priori* confidence whether Alice is friends with Bob. The goal of this privacy analysis is to understand whether the use of OUTOPIA is going to increase the server's confidence with regards to whether Alice is friends with Bob. Note that nobody can decrease the server's confidence of its knowledge of whether Alice is friends with Bob (the server can always ignore this information and resort to its *a priori* higher confidence). The server's goal is to capture more information to increase its *a posteriori* probability.

What can the server learn from dead-drops? In OUTOPIA, the disclosure probability DP is $1/B$ (B: the size of the bucket), since the server observes that Alice is accessing bucket B_{Alice_Bob}, which has B dead-drops. (i) If the *a priori* probability that the server had computed was less than $1/B$ (say $1/2B$), then the server will have gained some knowledge: the server increased its confidence that Alice is friends with Bob from $1/2B$ to $1/B$. In the extreme case where the bucket size is only one (i.e., $B = 1$), then the *a posteriori* probability is 1, and thus the server is 100% sure that Alice is friends with Bob. One the contrary, (ii) if the server's *a priori* probability was higher than $1/B$ (say it was $2/B$), then OUTOPIA did not really provide any helpful information to the server: it will ignore the fact that Alice is accessing Bob's bucket and stay with its *a priori* probability of $2/B$. Given that Bob retrieved the bucket that contains Alice's dead-drop, the server may try to increase its confidence that Alice is friends with Bob as follows: Let us assume that the server has an *a priori* estimate[2] of whether Alice is friends with each and every member (say i) of the bucket B_{Alice_Bob}, thus p_i: the probability that Bob is friends with the member who updates dead-drop i. In the best case (for the server) this estimate can be equal to "1" (i.e., the server is certain that Alice is friends with Bob) and in the worst case (for the server) the same value is much much lower. Based on the fact that Alice is accessing the bucket, which contains Bob's

[2]We make no assumptions where the server got this knowledge from.

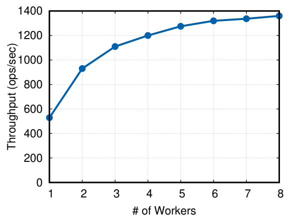

Fig. 6: Throughput of a bucket deltas read operation. The performance does level off after a sufficient number of clients hitting the server.

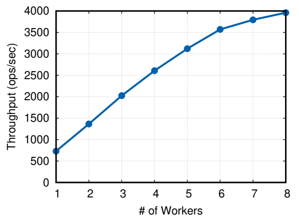

Fig. 7: Throughput of a dead-drop update operation. More than 4,000 ops/sec are achieved before the performance starts leveling off.

dead-drop, the server can compute an updated estimate of whether Bob is friends with Alice as follows:

- $\underline{Abob}$: $Alice\ is\ friends\ with\ Bob$
- $\underline{Abucket}$: $Alice\ accessed\ the\ Bucket$
- $\underline{Asome}$: $Alice\ is\ friends\ with\ someone\ in\ the\ Bucket$

$$p(Abob|Abucket) = p(Abucket|Abob) \times \frac{p_{Alice\_Bob}}{p(Asome)} \quad (2)$$

If Bob is friends with Alice, then obviously he will access the Bucket with probability 1: $p(Abucket|Abob) = 1$. The adversary now only needs to compute the probability $p(Asome)$: if the size B of the bucket is large and if we assume that the average probability of Alice being friend with anyone is $\overline{p}$, then $p(Asome) \approx B\overline{p}$. Thus, the updated probability is:

$$p(Abob|Abucket) = \frac{p_{Alice\_Bob}}{B\overline{p}}$$

Note that this updated probability will be larger than the *a priori* estimation p_{Alice_Bob}, if $p_{Alice\_Bob} < \frac{p_{Alice\_Bob}}{B\overline{p}}$ or if $B\overline{p} < 1$ which is true when: $B < 1/\overline{p}$. The above equation is a very powerful, yet elegant way to estimate what should be an appropriate bucket size: the bucket is large enough when the bucket size $B > 1/\overline{p}$.

5 EXPERIMENTAL EVALUATION

Implementation: To assess the feasibility and effectiveness of OUTOPIA, we develop a prototype of our system. We build two processes: a client and a server. The server module runs on an Apache web server listening for the `add()`, `update()`, and `get()` requests of the clients. Our API consists of only four different calls:

- `add_dead_drop(deadropID, userID, encrypted_message)`, which adds a new dead-drop in the system and returns the ID of bucket the dead-drop was placed in and its relative position (B_i, I_i) within this bucket,
- `update(deadropID, encrypted_message)` to update the value of a specific dead-drop of the system,
- `getBucket(`B_i`)` to request from the directory server a specific bucket, and finally
- `getBucketDelta(`B_i, TB_i`)` to request for possible dead-drop deltas of bucket B_i from time TB_i and after.

To implement the necessary cryptographic functions, we use the OpenSSL library: SHA-256 for secure hashing and the hybrid cryptosystem of Elliptic Curve Integrated Encryption Scheme (ECIES) [33] for public key encryption using 256-bit keys. ECIES allows us to produce relatively small ciphertexts while providing an adequate security level of 112 bits [34].

Operation	Execution Time
Read a bucket (the bucket contains 1,000 dead-drops)	4.26 ms
Read a bucket Delta (one dead-drop of the 1,000 is updated)	1.70 ms
Update a dead-drop in a bucket of 1,000 dead-drops	1.27 ms

Table 3. Execution time per operation. Alice reads the entire bucket only once, and this operation takes 4.26 ms; such a low latency is unable to affect her user experience. After that, Alice reads periodically bucket Deltas in 1.70 ms.

5.1 System Performance

Next, we set out to evaluate the system performance of our prototype. Our experimental infrastructure consists of two desktops (a client and a server) connected with Gigabit Ethernet. The client is equipped with a Hyper-Threading 6-core Intel Xeon E5-2620 processor operating at 2.00GHz, 15MB SmartCache, and 8GB of RAM. The server is equipped with a Hyper-Threading 12-core Intel Xeon E5-2697 v2 at 2.70GHz, 30MB SmartCache, and 32GB RAM.

Per Operation Performance: We measure how much time it takes Alice to find the IP address of Bob. That is, Alice asks OUTOPIA server for the bucket that contains Bob's dead-drop.

Reading an entire bucket: In our experiments we employ a (single-threaded) client querying the server for a particular bucket.We repeat each experiment 100 times and report averages. As seen in Table 3 it takes 4.26 ms on average to read an entire bucket (1000 dead-drops) which is practically unnoticeable. This includes the time it takes for the client to (i) fetch the proper bucket, (ii) filter the meaningful dead-drops and (iii) decrypt their contents.

Reading delta: We measure the execution time of the *read delta operation*: that is, the time it takes to retrieve the changes of the previously fetched bucket. Assuming that the delta regards the minimum possible change (one dead-drop changed), we see in Table 3 the time it takes to retrieve such delta is about 1.70 ms (2.5× less than reading the entire dead-drop). This relatively small difference is due to the fact that this time is dominated by connection establishing overheads (e.g., TCP/IP and TLS set up times).

Update dead-drop: We measure how much time it takes for Bob to update his dead-drop. As we see in Table 3 the time for an *update operation* is as low as 1.27 ms. Again, this time is dominated by socket and TLS set up times, since the amount of data that the client transfers is relatively small: 64 bytes per dead-drop.

Scalability: As a next step, we set out to explore the capabilities of a server running OUTOPIA and specifically how this scales with an increasing load. Thus, we measure the lower bound of throughput for a single directory server by hitting it with an increasing number of workers (that simulate users), each opening as many connections as are possible. Figure 5 presents the throughput of the server in bucket read operations as a function of the number of workers. Overall, a single directory server process was able to successfully serve more than 1,200 operations per second.

In Figure 6 we plot the results after measuring the throughput of the server in delta read operations. We see that for deltas, OUTOPIA can perform 1,400 operations per second. Similarly, in Figure 7 we present the number of dead-drop update operations per second. Since the updates are lightweight for the server and the client, the process can easily sustain over 4,000 operations per second. Overall, and considering that OUTOPIA can utilise as many servers as buckets, we see that it can easily scale to millions or even billions of users.

6 DISCUSSION

Inactive Users: There are cases, where users may stop updating their dead-drops and become inactive. Thus, one may say that since no querying users are actually interested in idle dead-drops, the server can increase the disclosure probability of the rest of the dead-drops in the particular bucket. Fortunately though, there are also users that are not very mobile and thus rarely (if not never) need to update their dead-drops. The friends of these non-mobile users will continue to fetch deltas for this bucket. So, the server cannot distinguish a non-mobile user from an inactive user and thus cannot increase the disclosure probability of the dead-drops.

Bidirectional friendships: So far, we described the case that Alice wants to talk to Bob, but friendships are usually bidirectional and Bob may want to talk to Alice, too. In these cases, the server may have a higher probability in estimating that Alice and Bob are friends: e.g., assume that when Bob wants to share his IP address with Alice, he uses a dead-drop in bucket B_1 and when Alice wants to share her own with Bob, she uses bucket B_2. If the server observes only bucket B_1 or only bucket B_2 then it has little information on whether Alice is friends with Bob. However, the server may *combine* the information from B_1 and B_2 as follows: Bob updates a dead-drop in B_1 and reads all dead-drop in B_2. Thus, Bob is contained in the intersection of two sets: (i) the set of clients who update B_1 and (ii) the set of clients who read B_2 – let us call this intersection B_{all}. Similarly Alice is contained in the intersection of two sets: (i) the set of clients who update B_2 and (ii) the set of clients who read B_1 – let us call this intersection A_{all}. By observing both B_1 and B_2, the server knows that each one of the people in B_{all} are friends with at least one of the members A_{all}. If the sets B_{all} and A_{all} are very small (say singletons), the server is able to conclude with a very high probability that Alice is friends with Bob. To prevent the server from reaching this conclusion, we need to ensure that B_1 and B_2 are located in different servers which do not collude (similar to other approaches [23]). Fortunately, this is not difficult since OUTOPIA is inherently parallel and can employ several different servers – as many as one server per bucket. Thus, Alice may easily choose a server different from the one Bob used to store his dead-drop.

Freshness of information: Alice requests for the bucket every T seconds. This means that if Bob changes his IP address Alice may take as many as T seconds to get the update. Depending on T, this may be too long for Alice. In our system, we envision a value of T in the range of several seconds to a few minutes. Figure 3 suggests that if the epoch duration T is 4 minutes, the generated traffic is about 1 Kbps. If the epoch duration drops down to 0.5 minute, the generated traffic is about 4 Kbps. So an epoch size of around 1 minute sounds reasonably small to keep the generated traffic to small numbers. We understand that one might say that waiting for one minute or even 30 seconds for Alice to talk to Bob might be too much. We must make clear that when Alice wants to talk to Bob, she does not wait for 30 seconds to find Bob's IP address. Alice already knows Bob's IP address (every 30 seconds she downloads the IP addresses of all her friends) so when she wants to talk to Bob there is no delay.

7 RELATED WORK

Riposte [20] leverages PIR and scales to a few million users under the assumptions that (i) only a small fraction of users write to the database, and (ii) epochs last several hours (few hundred writes per second). Although Riposte can protect the social associations, it relies on broadcasting all messages to all users thus imposing significantly high communication costs. Dissent [21] and Herbivore [35] adopt the similar broadcast nature and their scalability is limited to broadcasting groups of up to 5K users each. In [19], authors propose Apres, the approach behind Drac [36]. Apres assumes a dedicated server, which serves as a hub able to store and forward messages. Apres requires the use of Tor to access the server anonymously. Thus, whenever Alice wants to connect with Bob, she connects via Tor to the server, and adds her message in an *implicit address*, created just for them (similar to our dead-drop). PROUD [27] is a privacy preserving User Discovery service that leverages the existing DNS to distribute cryptographically secure user-to-IP mappings. Contrary to our work, PROUD assumes two non-colluding type of DNS nodes for user record registering and resolving. Loopix [37] is an anonymous communication system that provides bi-directional anonymity and unobservability. It leverages Poisson mixing and constantly sends drop cover messages to random receives. Overall message latency is on the order of seconds. Similar to our approach, Loopix assumes that a fraction of the mix/provider relays are honest.

Vuvuzela [18] is a system for private communication under heavy surveillance. It uses chains of servers that use onions and cover traffic (based on the principles of differential privacy) to conceal the users' message exchanges. Vuvuzela along with its enhancement Alpenhorn [38] can preserve the privacy of the user's social graph. Its heavyweight approach requires the users to be constantly online and participate in every round by sending *no-op* messages even during their idle times. Apparently this results to a significant bandwidth requirement per user (around 12 KBytes/sec – or 12 GBytes/sec in total for 1M users). Pung [22] is a key-value store to provide private communication based on a computational PIR model. To improve its performance, Pung uses a probabilistic multi-retrieval scheme, thus allowing its server to efficiently process multiple retrievals from the same user. Although this scheme allows Pung to reduce computational costs by up to 11×, it significantly increases its network costs, thus rendering its applicability as questionable.

8 CONCLUSION

In this paper, we decouple User Discovery from the messaging apps by proposing *OUTOPIA*: an independent third-party discovery service to provide privacy-preserving User Discovery without revealing who is socially connected with whom. Our approach provides *measurable privacy* individually to every social association of the user. Preliminary performance results suggest that OUTOPIA is scalable for the increasing number of users and users are able to discover each other in no more than a few milliseconds, while generating a negligible traffic overall. We envision OUTOPIA as a multi-purpose discovery system which can be integrated in several other applications beyond IP addresses, thus allowing Bob to privately share e.g., his telephone number, current geolocation or Tor onion service.

Acknowledgements: This project received funding from the EU H2020 Research and Innovation programme under grant agreement No 830929 (CyberSec4Europe). These results reflect only the authors' view and the Commission is not responsible for any use that may be made of the information it contains.

REFERENCES

[1] Whatsapp. Whatsapp encryption overview - technical white paper. https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf.
[2] Facebook Inc. Messenger secret conversations - technical whitepaper. https://fbnewsroomus.files.wordpress.com/2016/07/secret_conversations_whitepaper-1.pdf.
[3] Apple Inc. iOS Security Guide - White Paper. https://apple.com/business/docs/iOS_Security_Guide.pdf.
[4] Christoph Hagen, Christian Weinert, Christoph Sendner, Alexandra Dmitrienko, and Thomas Schneider. All the numbers are us: Large-scale abuse of contact discovery in mobile messengers. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS'20, 2020.
[5] Alan Mislove, Bimal Viswanath, Krishna P. Gummadi, and Peter Druschel. You are who you know: Inferring user profiles in online social networks. In *Proceedings of the Third ACM International Conference on Web Search and Data Mining*, WSDM, 2010.
[6] Elizabeth Denham. Why we should worry about whatsapp accessing our personal information. https://www.theguardian.com/commentisfree/2016/nov/10/whatsapp-access-personal-information-privacy-facebook-consumers-information-commission, 2016.
[7] Andrea Peterson. Bankrupt radioshack wants to sell off user data. but the bigger risk is if a facebook or google goes bust. https://www.washingtonpost.com/news/the-switch/wp/2015/03/26/bankrupt-radioshack-wants-to-sell-off-user-data-but-the-bigger-risk-is-if-a-facebook-or-google-goes-bust/, 2015.
[8] Yael Grauer. What are data brokers, and why are they scooping up information about you? https://motherboard.vice.com/en_us/article/bjpx3w/what-are-data-brokers-and-how-to-stop-my-private-data-collection.
[9] Federal Trade Commission. FTC sues failed website, toysmart.com, for deceptively offering for sale personal information of website visitors. https://www.ftc.gov/news-events/press-releases/2000/07/ftc-sues-failed-website-toysmartcom-deceptively-offering-sale, 2000.
[10] Parmy Olson. Facebook closes 19 billion whatsapp deal. www.forbes.com/sites/parmyolson/2014/10/06/facebook-closes-19-billion-whatsapp-deal.
[11] Greg Mahlich Robert-Jan Bartunek, Philip Blenkinsop. Eu fines facebook 110 million euros over whatsapp deal. http://www.reuters.com/article/us-eu-facebook-antitrust-idUSKCN18E0LA, 2017.
[12] Julie Tate Barton Gellman and Ashkan Soltani. In nsa-intercepted data, those not targeted far outnumber the foreigners who are. https://www.washingtonpost.com/world/national-security/in-nsa-intercepted-data-those-not-targeted-far-outnumber-the-foreigners-who-are/2014/07/05/8139adf8-045a-11e4-8572-4b1b969b6322_story.html, 2014.
[13] Andy Greenberg. Google hands over user data for 94% of u.s. law enforcement requests. https://www.forbes.com/sites/andygreenberg/2011/06/27/google-hands-over-user-data-for-94-of-law-enforcement-requests/#7f8f1dc62f89.
[14] Joon Ian Wong. Here's how often apple, google, and others handed over data when the us government asked for it. https://qz.com/620423/heres-how-often-apple-google-and-others-handed-over-data-when-the-us-government-asked-for-it/.
[15] Carter Jernigan and Behram FT Mistree. Gaydar: Facebook friendships expose sexual orientation. *First Monday*, 14(10), 2009.
[16] Moxie Marlinspike. Technology preview: Private contact discovery for signal. https://signal.org/blog/private-contact-discovery/, 2017.
[17] Alma Whitten and J. D. Tygar. Why johnny can't encrypt: A usability evaluation of PGP 5.0. In *8th USENIX Security Symposium (USENIX Security 99)*, Washington, D.C., August 1999. USENIX Association.
[18] Jelle Van Den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP'15, 2015.
[19] Ben Laurie. Apres-a system for anonymous presence. Technical Report, 2004.
[20] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. *arXiv preprint arXiv:1503.06115*, 2015.
[21] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Dissent in numbers: Making strong anonymity scale. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2012.
[22] Sebastian Angel and Srinath Setty. Unobservable communication over fully untrusted infrastructure. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, 2016.
[23] Nikita Borisov, George Danezis, and Ian Goldberg. Dp5: A private presence service. In *Proceedings on Privacy Enhancing Technologies*, PETs'15, 2015.
[24] Raymond Cheng, William Scott, Elisaweta Masserova, Irene Zhang, Vipul Goyal, Thomas Anderson, Arvind Krishnamurthy, and Bryan Parno. Talek: Private group messaging with hidden access patterns. In *Annual Computer Security Applications Conference*, pages 84–99, 2020.
[25] Ania M Piotrowska, Jamie Hayes, Nethanel Gelernter, George Danezis, and Amir Herzberg. Anonotify: A private notification service. *IACR Cryptol. ePrint Arch.*, 2016:466, 2016.
[26] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*, FCC'95. IEEE, 1995.
[27] Panagiotis Papadopoulos, Antonios A Chariton, Elias Athanasopoulos, and Evangelos P Markatos. Where's wally? how to privately discover your friends on the internet. In *Proceedings of 13th ACM ASIA Conference on Information, Computer and Communications Security*, ASIACCS'18, 2018.
[28] David McCandless. World's biggest data breaches. http://www.informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks/, 2015.
[29] Brian Krebs. Privacy 101: Skype leaks your location. http://krebsonsecurity.com/2013/03/privacy-101-skype-leaks-your-location/, 2013.
[30] Stevens Le Blond, Chao Zhang, Arnaud Legout, Keith Ross, and Walid Dabbous. I know where you are and what you are sharing: exploiting p2p communications to invade users' privacy. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, IMC'11, pages 45–60. ACM, 2011.
[31] Panagiotis Papadopoulos, Antonis Papadogiannakis, Michalis Polychronakis, Apostolis Zarras, Thorsten Holz, and Evangelos P Markatos. K-subscription: Privacy-preserving microblogging browsing through obfuscation. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC'13, 2013.
[32] Panagiotis Papadopoulos, Antonis Papadogiannakis, Michalis Polychronakis, and Evangelos P. Markatos. Is privacy possible without anonymity? the case for microblogging services. In *Proceedings of the 12th European Workshop on Systems Security*, EuroSec, 2019.
[33] Victor Shoup. A proposal for an iso standard for public key encryption (version 2.1). *IACR E-Print Archive, 112*, 2001.
[34] V Gayoso Martínez, L Hernández Encinas, and C Sánchez Ávila. A survey of the elliptic curve integrated encryption scheme. *ratio*, 2010.
[35] Sharad Goel, Mark Robson, Milo Polte, and Emin Sirer. Herbivore: A scalable and efficient protocol for anonymous communication. Technical report, Cornell University, 2003.
[36] George Danezis, Claudia Diaz, Carmela Troncoso, and Ben Laurie. Drac: An architecture for anonymous low-volume communications. In *Privacy Enhancing Technologies*, PETs'10, 2010.
[37] Ania M Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. The loopix anonymity system. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 1199–1216, 2017.
[38] David Lazar and Nickolai Zeldovich. Alpenhorn: Bootstrapping secure communication without leaking metadata. In *In proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'16, 2016.